Computing (2012) 94:857–885
DOI 10.1007/s00607-012-0204-1

SEON: a pyramid of ontologies for software evolution and its applications

Michael Würsch · Giacomo Ghezzi · Matthias Hert · Gerald Reif · Harald C. Gall

Received: 22 February 2012 / Accepted: 27 June 2012 / Published online: 14 July 2012

Abstract The Semantic Web provides a standardized, well-established framework to define and work with ontologies. It is especially apt for machine processing. However, researchers in the field of software evolution have not really taken advantage of that so far. In this paper, we address the potential of representing software evolution knowledge with ontologies and Semantic Web technology, such as Linked Data and automated reasoning. We present SEON, a pyramid of ontologies for software evolution, which describes stakeholders, their activities, artifacts they create, and the relations among all of them. We show the use of evolution-specific ontologies for establishing a shared taxonomy of software analysis services, for defining extensible metamodels, for explicitly describing relationships among artifacts, and for linking data such as code structures, issues (change requests), bugs, and basically any changes made to a system over time. For validation, we discuss three different approaches, which are backed by SEON and enable semantically enriched software evolution analysis.

This work was supported by the Swiss National Science Foundation as part of the ProDoc "Enterprise Computing" (PDFMP2-122969) and the "Systems of Systems Analysis" (200020_132175) projects.

M. Würsch (✉) · G. Ghezzi · M. Hert · G. Reif · H. C. Gall
Department of Informatics, University of Zurich,
Binzmühlestrasse 14, 8050 Zurich, Switzerland
e-mail: wuersch@ifi.uzh.ch

G. Ghezzi
e-mail: ghezzi@ifi.uzh.ch

M. Hert
e-mail: hert@ifi.uzh.ch

G. Reif
e-mail: reif@ifi.uzh.ch

H. C. Gall
e-mail: gall@ifi.uzh.ch

These techniques have been fully implemented as tools and cover software analysis with web services, a natural language query interface for developers, and large-scale software visualization.

Keywords Software evolution · Semantic Web · Ontologies

Mathematics Subject Classification 68U01 · 68U35

1 Introduction

Scientia potentia est. Knowledge is power. For millennia this maxim has been valid, and will likely remain so in the future—even in an age of information overload, where the entire humankind produces roughly two zettabytes data a year.[1]

This also holds for the domain of software engineering, where even small development teams accumulate gigabytes of interdependent artifacts over the years. They are stored in software repositories, such as version control systems, issue trackers, but also in Wikis, and even mailing lists. Understanding what factors distinguish successful development projects from others is key to improve the quality of software systems. Distilling the knowledge of best practices from random noise found in a software repository is what the field of software evolution research and mining software repositories aims for.

But data is not necessarily information, and information not necessarily knowledge. Successful differentiation requires understanding of data semantics and interpretation. The obvious solution to this dichotomy is that machines and humans form a joint-venture: humans define the semantics and machines bring in their computational power for the advent of the next generation of software evolution support tools. The Semantic Web provides the instruments to achieve such a synergy; ontologies created by human beings represent knowledge and give semantic meaning to raw data so that machines can automatically process and exchange it. Reasoners make implicit knowledge explicit by inferring relations that were previously missing. Interestingly, these technologies yet struggle to find a wide adoption in the field of software evolution research, whereas, for example in life sciences, many applications have demonstrated the value of the Semantic Web for processing and sharing large corpora of information (e.g., in [41]).

In this paper, we pursue the research question, how we can adequately describe software evolution knowledge by means of ontologies. This includes knowledge about stakeholders, activities, artifacts, and the relations among all of them. The ultimate goal is to provide software engineers with effective tool-support for managing software systems over their entire life-cycle.

The contributions of our paper are threefold:

1. We critically reflect on the potential that the Semantic Web yields for software evolution. In particular, we show four characteristics that are most beneficial for the

[1] According to the study "Digital Universe: Extracting Value from Chaos" by IDC, humans created 1.8 zettabytes data in 2011. This value is estimated to double every two years.

field: shared taxonomies, extensible meta-models, explicit relations, and Linked Data.
2. We present SEON, our family of software evolution ontologies. These ontologies describe knowledge on multiple levels of abstraction ranging from code structures up to stakeholder activities.
3. We describe three semantics-aware tools that make extensive use of SEON and help developers in dealing with large amounts of software evolution data: software analysis with web services, a natural language query interface for developers, and large-scale software visualization. All three of them have been fully implemented for a proof-of-concept.

In the remainder of this paper, we will describe the potential of Semantic Web technology for dealing with software evolution.

In Sect. 2, we give a brief overview on the Semantic Web and related technologies, before we discuss in Sect. 3 the advances they can bring to the field of software evolution research. We also address a set of general challenges yet to be solved before the full potential of Semantic Web-enabled approaches can be realized.

At the core of this paper is SEON, our pyramid of ontologies for software evolution, which is described in Sect. 4. These ontologies provide a taxonomy to share software evolution data of various abstraction levels across the boundaries of different tools and organizations.

In Sect. 5, we describe three different applications of SEON from three distinct domains to showcase the utility and versatility of ontologies in the context of software evolution research. A selection of other ontology-driven approaches in the field of software engineering is discussed in Sect. 6. In Sect. 7, we conclude the paper.

2 The Semantic Web in a nutshell

Berners-Lee et al. [4] define the Semantic Web as "an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation".

Despite its origins, the Semantic Web is not limited to annotating webpages with meta-data. Virtually any piece of knowledge can be described in a computer-processable way by defining an ontology for the domain of discourse. An ontology formally describes the concepts (classes) found in a particular domain, as well as the relationships between these concepts, and the attributes used to describe them [22]. For example, in the domain of software evolution, we define concepts, such as *User*, *Developer*, *Bug*, or *Java Class*; relationships, such as *reports bug*, *resolves bug*, or *affects Java Class*; and attributes, such as *email address of developer*, *resolution date of bug*, *severity of bug*, etc.

Since the Semantic Web describes knowledge based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first place. The data representation format no longer needs to be custom-tailored to a specific task, but can be re-used later.

Researchers and practitioners came up with a number of standards, W3C recommendations, development frameworks, APIs, and databases to pursue the vision of the

Semantic Web. The Resource Description Framework (RDF) [40] is the data-model for representing meta-data in the Semantic Web. The RDF data-model formalizes meta-data based on *subject–predicate–object* triples, so called RDF statements. RDF triples are used to make a statement about a resource of the real world. A resource can be almost anything: a project, a bug report, a person, a Web page, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [3].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model, information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [49] can be used to query such RDF graphs.

RDF itself is domain-independent in that no assumptions about a particular domain of discourse are made. It is up to the users to define specific ontologies in an ontology definition language, such as the Web Ontology Language (OWL) [10]. OWL enables the use of description logic (DL) expressions to further describe the relationships between classes and to restrict the use of properties [47]. For example, two classes can be declared to be disjoint, new classes can be built as the union/intersection of others, or the cardinality of a property can be restricted to define how often a property can be applied to an instance of a class. OWL can describe both uniformly, data schema and instance data.

In addition to the W3C recommendations, the Semantic Web community developed tools to process RDF meta-data. Jena[2] emerged from the *HP Labs Semantic Web Program* and recently became an Apache incubator project. It is a Java framework for building applications for the Semantic Web and provides a programmatic environment for RDF and OWL. Reasoners, e.g., Pellet[3] or HermiT,[4] infer logical consequences from a set of asserted facts or axioms. RDF databases, such as Sesame[5] or Virtuoso,[6] store RDF triples and can be queried with SPARQL.

3 The potential of ontologies in software evolution research

Over the last decade, software evolution research brought up various tools that help engineers to better deal with large, ever-changing legacy systems. In [60] it was argued that most of these tools use proprietary data formats to store their artifacts, which hampers tool-interoperability. Furthermore, querying software evolution knowledge is difficult, especially when queries span across different domains. Queries such as "In which release was this bug fixed and which source code modifications where done to fix it?" involve several domains (i.e., static source code, version control, issue tracking), something which is not originally supported by common software repositories.

[2] http://incubator.apache.org/jena/.

[3] http://clarkparsia.com/pellet/.

[4] http://hermit-reasoner.com/.

[5] http://www.openrdf.org/.

[6] http://virtuoso.openlinksw.com/.

The *Mining Software Repositories*[7] community tackled this issue by mirroring software artifacts from various sources in a central (relational) database [9]. This gave rise to numerous experiments where researchers successfully mined such databases for interesting patterns (see [34] for an overview; specific examples can be found in [7,16,19,52]). Unfortunately, such a central database imposes a universal data schema onto all contributing tools, turning the software repository into a rigid and inflexible monolith.

Semantic Web technology has been designed as a solution to such integration problems. In the following, we briefly revisit the characteristics of the Semantic Web that we identified in in our previous work to be most beneficial for the field of software evolution research.

3.1 Establishing a shared taxonomy of software evolution

One of the critical design aspects when building a knowledge base is to define a meta-model that describes the knowledge in an adequate level of detail. To share data among different tools, they need to understand the same vocabulary.

In practice, there are a number of general-purpose meta-models in software engineering, such as the Dagstuhl Middle Metamodel (DMM) [43], as well as more specific ones, e.g., for source code. Many of them define the same concepts, but name them differently. The *C++ Data Model* [8] of Chen and the *FAMOOS Information Exchange Model (FAMIX)* of Tichelaar et al. [54] can both be used to describe source code written in C++. Although they share many commonalities, tools written to work on FAMIX cannot process instances of Chen's model and vice versa, e.g., to replicate experiments. Further, meta-models are often implemented in terms of a relational database schema. Exchanging schemata among different databases, however, is relatively inconvenient, due to vendor-specific implementations of data definition languages. Instead, and despite the advent of specialized exchange formats, such as RSF [44], XMI [46], or GXL [57], data is often serialized into plain XML or a comma separated value (csv) format. These formats are not semantics-preserving and therefore of limited use.

While relational database schemata are hardly ever exchanged, ontologies were explicitly designed to be shared. They can be serialized using the RDF/XML standard and exchanged without loss of data semantics. In Sect. 4, we propose our set of ontologies that provide a taxonomy for important concepts in the domain of software evolution. With the approach described in Sect. 5.1, we demonstrate how such taxonomy fosters interoperability between an entire ecosystem of software services.

3.2 Defining extensible meta-models

Especially in a research context, meta-models tend to evolve constantly. Therefore, they need to be designed to be extensible. For example, adding data about additional software artifacts should be straight-forward and possible without breaking applications that rely on the original model.

[7] http://www.msrconf.org/.

When meta-models are extended, this usually enforces database schema changes—a time consuming operation, as the whole repository and all database keys have to be reorganized. Chances are more than likely that existing applications directly accessing the database will break in such a case.

Designing ontologies is comparable to designing Entity-Relationship or UML models. The result is a data schema. In the Semantic Web, however, the schema itself is described in terms of RDF triples, making it more flexible to changes than the relational one. No distinction between data and ontology is necessary, as both are simply additions or deletions of triples. It is therefore unproblematic to add more ontologies and to specialize existing concepts and properties by deriving sub-concepts and sub-properties.

In Sect. 5.2 we present a query approach that especially benefits from the extensibility of ontologies, as well as from the fact that data and meta-data are represented uniformly. Our query system analyses both, the data and meta-data and uses the results to guide developers in composing and executing queries related to program comprehension tasks. When we add new ontologies to SEON, our query system is able to deal with this additional knowledge without requiring us to change a single line of code.

3.3 Making relations explicit

There is no consistent way to get the meaning of a relation in relational databases. In fact, a query can join tables by any columns, which match by datatype—without any check on the semantics. While humans can often guess the meaning of a relation, computers can not. They need to be supplied with additional information. It is therefore necessary to encode a significant amount of implicit knowledge into applications to make use of the data. To search in an existing repository, or to build an own tool on top of it, researchers need to be aware of, and understand this implicit semantics.

The SPARQL query language allows one to query explicitly for relations among resources. Such queries are impossible in the relational and in the object-oriented paradigm unless relationships are explicitly mapped to tables or, in the case of object-orientation, modeled as association classes. The latter, however, can make them difficult to distinguish from "real" classes. Given the high importance of relationships in software evolution, it is preferable to model them as first class objects—which is exactly what the Semantic Web does.

The importance of this aspect is emphasized in Sect. 5.3. There we introduce our recommender tool, which depends on the explicit semantics of ontologies. Given a set of data, it searches for certain types of individuals, as well as for their relations, to recommend appropriate visualizations.

3.4 Linked software evolution data

With only relational database technology, synergies between research tools are hard to exploit. For example, we cannot simply establish connections between data stored in two different software repositories, such as a version control system and an issue

tracker. The reason for this is that it is impossible to set a link from one repository to another—relations are local, not universal. Cross-domain queries spanning multiple repositories are impossible.

One of the driving forces behind the Semantic Web is the basic assumption that data becomes more useful the more it is interlinked with other data. The simple but powerful concept of statements represented by triples of URIs can be used to build an internet-scale graph of information because it makes it possible to link and query data that is stored in different locations.

The software analysis services described in Sect. 5.1 manage data based on these principles. URIs are assigned to every artifact analyzed and all the results generated. These URIs are de-referenceable over the Web and allow services to request from other (remote) services information about resources on an as-needed basis. Like that, the software analysis services already operate on a global graph of software evolution data today.

In the next section, we describe SEON, an ontological description of the domain of software evolution. It exploits the characteristics of the Semantic Web mentioned above to support a wide range of semantics-aware applications.

4 SEON: a pyramid of ontologies for software evolution

The acronym SEON stands for *Software Evolution ONtologies* and represents our attempt to formally describe knowledge from the domain of software evolution analysis & mining software repositories. However, in contrast to many other existing ontologies, we did not aim to capture as much of the domain under discourse as possible. Instead, we originally incorporated only a limited set of concrete concepts and extended the ontologies solely when it was actually required by a particular analysis or by a tool that we had already built or used. Three of these tools are detailed in Sect. 5. We then followed a bottom-up approach and, from these very concrete concepts, iteratively added abstractions and extended our ontologies. This process is briefly described in Sect. 4.6.

Figure 1 presents an overview of the different layers of SEON. The most distinguishing feature is, compared to other ontologies related to the domain of software evolution, the strict organization into different levels of abstraction. In the following, we explain each of the layers that comprise our pyramid of ontologies. We focus on a few examples but do not provide a detailed description for every concept defined in SEON in this paper. Instead, we explain the general structure of our ontology pyramid and the rationale behind its design. Interested readers are invited to browse our OWL definitions online.[8] At the end of this section, we give an example on how the different layers can be used in conjunction with each other to describe knowledge in a concrete analysis scenario, namely the analysis of the evolution of code clones in a software system.

[8] http://www.se-on.org/ontologies/.

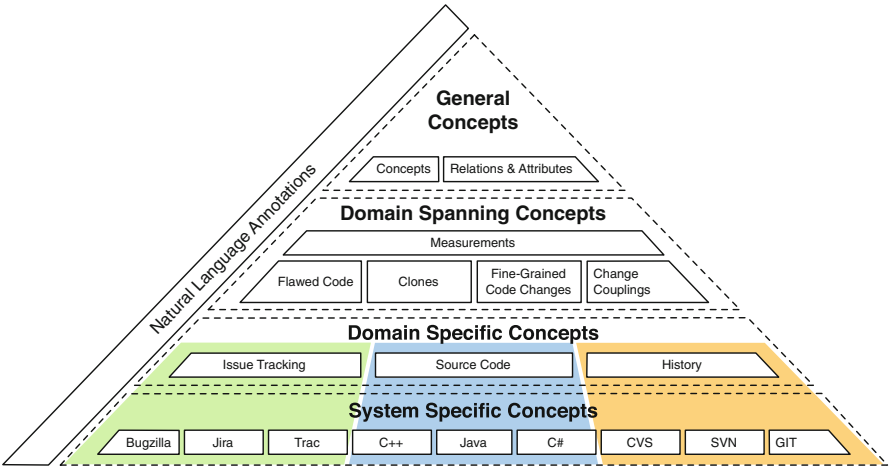

Fig. 1 The software evolution ontology pyramid

4.1 General concepts

The pyramidion, i.e., the top layer, is comprised of domain-independent or general concepts, the attributes that describe them, and the relations between the concepts.

Concepts are modeled by OWL classes. Instances of classes are OWL individuals. OWL datatype properties represent attributes, and OWL object properties the relations between concepts. The first ones link individuals to data values, whereas the latter ones link individuals to individuals. To better differentiate terms, we underline OWL classes in this section. A dotted underline denotes individuals and a dashed underline is used for properties.

Classes in the top-layer relate to concepts omnipresent in software evolution. Examples are Activity, Stakeholder, or File. We also defined a set of datatype properties for generic attributes, such as hasSize or createdOn. They are domain-independent; files, program execution stack traces, but also project teams have a size. Similarly, requirement documents, bug reports, or mailing list entries are attributed a creation date.

SEON also defines a more extensive set of domain-independent object properties. These properties are fundamental to many applications, as relations between "things" are paramount for most analyses in software evolution. On this level of abstraction, there is for example the concept of authorship, as any artifact in software evolution has one or several authors, denoted by the object property hasAuthor. Our ontology also has an object property called dependsOn that generalizes many different relations in the software evolution domain.[9] Specializations of dependsOn therefore can range from other domain-independent properties, such as hierarchical relationships (i.e., a *parent-child* relationship), to more domain-specific ones, e.g., dependencies between requirements or static source code dependencies. Such domain-specific properties, however, are specified in lower layers of SEON, as sub-properties of higher-level ones.

[9] The concept of inheritance in OWL goes further than in object-orientation. Not only OWL classes can inherit from other classes, but also OWL object and data properties can inherit from other properties.

Another domain-independent object property defines the abstract notion of similarity between two individuals. The concept of similarity, again, is universal. It applies to source code (a.k.a. "code clones"), as well as to issues (a.k.a. "bug duplicates") and many other artifacts. What "similar" actually means in a specific case, however, is then up to the fact extractors to decide when they instantiate SEON models.

What is the benefit of having defined the abstractions described above? First, we as human beings are comfortable with thinking in categories—this capability develops as early as within the first half year of our lives [29]. Categorization and taxonomizing things help us to understand the complex domain of software evolution. Second, as we will describe in the remainder of this paper, such abstractions enable us to build flexible, largely domain-independent tools to support many different facets of software evolution activities.

4.2 Domain-spanning concepts

The second-highest layer of SEON defines domain-spanning concepts. These concepts are less abstract than the general concepts. They describe knowledge that spans a limited number of subdomains, e.g., version control systems and source code in the case of our change coupling ontology. Change couplings describe implicit relationships between two or more software artifacts that frequently change together during evolution [2,18]. Other ontologies related to the version history of program code cover fine-grained source code changes and code clones. The ontology for fine-grained source code changes describes program modifications not only on a file level but also down to the statement level. It is based on the CHANGEDISTILLER meta-model of change types [17]. The code clone ontology is able to describe duplicated code and how it evolves over time. Similarly to the code clone ontology, our ontology about flawed code is concerned with quality attributes of source code. The ontology represents knowledge distilled from issue trackers and version control systems. It describes the bug history of files or modules, but also of individual classes or even methods in object-oriented programs. Furthermore, it covers *Design Disharmonies* [42] or, in other words, formalized design shortcomings found in source code, e.g., Brain Classes, Feature Envy, Shotgun Surgery, etc.

Another important concept is that of a Measurement. A sophisticated ontology for software measurement has been presented by Bertoa et al. [6]. SEON adapts some of the most important concepts identified by these authors, but we weigh simplicity over completeness by leaving out those that have not played a crucial role in our recent analyses.

A measurement is the act of measuring certain attributes of a software artifact or process; a Measure, or metric, is the approach taken to perform a measurement. Measures have a Unit, such as *number of bugs per line of code*. Measured values are expressed on a Scale, e.g., an ordinal or nominal scale. Information about units and scales can be used to perform conversions, for example, to compare the results of different measurements. While the abstract concepts are defined in the pyramidion, many primitive measures are domain-specific. Still we consider measurements to belong mainly to the layer of domain-spanning concepts. Primitive measures, such as *number of lines of code* and *number of closed bugs*, on their own are not very meaningful and need to be put into relation in order to derive a meaningful assessment

of a software system's health state. The most effective measurements therefore are based on derived measures [42]; they present an aggregation of values from different subdomains. The *number of bugs per class* is computed from values originating from the source code and the issue tracker, and the *level of class ownership* is derived from source code and commits to a version control system.

In summary, SEON's layer of domain-spanning concepts describes software evolution knowledge on the level of analyses and results, whereas the remaining two layers describe raw data, i.e., artifacts and meta-data directly retrieved from repositories.

4.3 Domain-specific concepts

The third layer is divided into different domains corresponding to important facets of the software evolution process, that is, among others, issue and version management. It includes a taxonomy for source code artifacts encountered in object-oriented programming. While the concepts defined in this layer are specific to a domain, they are independent of technology, vendor, and version. Each domain captures the commonalities shared among the many different issue trackers, object-oriented programming languages, or version control systems.

The majority of issue trackers are organized around Issues that can be divided into Bugs, FeatureRequests, and Improvements. Issues are reportedBy someone and assigned to a developer for fixing them. Object-oriented programming languages usually consist of Classes organized in some kind of Namespaces. Classes declare Members—Methods and Fields—and they can inherit from other classes. Developers modify files in resolving issues and commit them to a version control system resulting in a new Revision for these files. They organize their repository with respect to development streams into Branches and prepare from time to time a Release of the system under development. All these concepts—and many more—are formally defined in SEON. These definitions build a taxonomy that can be shared among researchers and practitioners, but also among machines.

Concepts do not necessarily need to be present in all of the systems that are abstracted by the domain-specific layer. The concept of, e.g., Mixins does not exist in Java but in other languages, such as Scala and Smalltalk. Defining this concept nonetheless is perfectly valid, as it is a common concept in object orientation. There will simply be no instances of such concepts if SEON is used to describe a software system written in Java or any other language that does not support them.

While devising the layer of domain-specific concepts, we maintained a bird's-eye view on commonly used technologies that are conceptually related, yet very different in implementation. Our goal was to distill some of the essentials of software evolution into a set of meta-models. These meta-models, however, are not static. They are destined to evolve, as the body of software engineering knowledge grows.

4.4 System-specific concepts

Whereas the third layer describes domain-specific concepts that apply to families of systems, the bottom layer defines system-specific concepts. It extends the knowledge

of the upper layers by concepts unique to certain programming languages, vendors, versions, or specific tool implementations. We aim to keep this layer as thin as possible while capturing relevant information beneficial for analyzing specific facets of the evolution of concrete programs. For some systems, we have barely seen the need to define specific concepts, without loosing crucial information. Other systems differ significantly from the baseline and require more system-specific knowledge.

One example for system-specifics is the severity of issues. While most modern issue trackers know the concept of severity to classify an issue, their concrete implementations vary quite substantially. The different levels of severity, as well as their naming, depends very much on the particular issue tracker and, in some cases, even on how it is configured by development teams. Still, the information is valuable, e.g., as input for machine learning algorithms when experimenting with automated bug triaging approaches [21]. Therefore we defined Severity in the layer of domain-specific concepts, but the individuals that represent the different levels of severity are covered in system-specific ontologies. System-specific parsers then extract this information and link individuals of Issue to the corresponding individuals of Severity.

4.5 Natural language annotations

The Semantic Web was not primarily devised for human beings consuming information. Instead its conception is that machines become capable of processing the knowledge of humans and there is usually additional effort of knowledge engineers needed to encode it in an adequate format.

Despite this machine-centric design, there are many occasions where humans need to interface with Semantic Web data. Therefore, we added a layer of natural language annotations to SEON. These annotations provide human-readable labels for all classes and properties. For individuals, we use RDF Schema labels (*rdfs:label*).

In particular, we defined the following custom annotations as subclasses of the OWL *AnnotationProperty*.[10] The most important three annotations in the natural language layer are:

- **phrase-s** adds singular synonyms to OWL classes and properties.
- **phrase-p** adds plural synonyms to OWL classes and properties.
- **explanation:** adds a human-readable description to OWL classes and properties

The encoding of the grammatical number of a synonym (*phrase-s* vs. *phrase-p* annotation) is important in order to correctly translate statements from OWL to natural language. The *explanation* annotation is very similar to the RDF Schema comment annotation (*rdfs:comment*) defined by the W3C, except that our annotation is explicitly meant to be shown in user interfaces to end-users (e.g., in tooltips), whereas *rdfs:comment* is also often used to document OWL classes and properties for knowledge engineers.

In Fig. 2, we show an excerpt of an RDF graph as an example of how we annotate our SEON ontologies with natural language. For the concept Developer, we added multiple

[10] http://www.w3.org/2002/07/owl#AnnotationProperty.

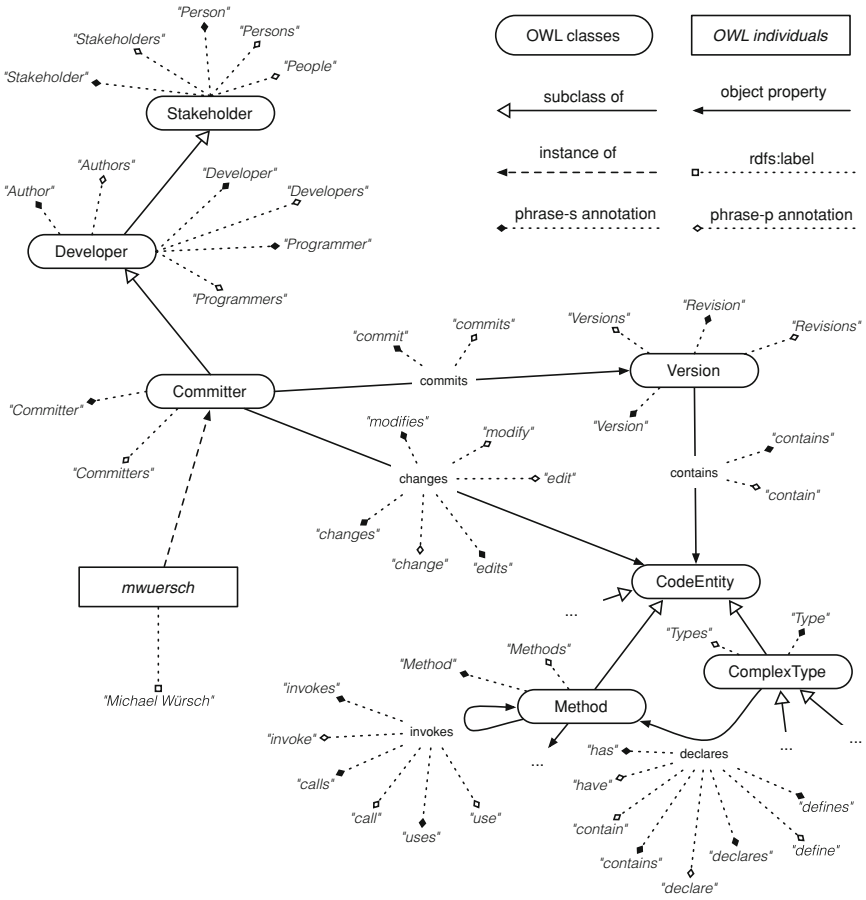

Fig. 2 RDF graph with natural language annotations

natural language representations, in particular the nouns *Author(s)*, *Developer(s)*, and *Programmer(s)*. The annotations from its super-concept Stakeholder—*Stakeholder(s)*, *Person(s)*, and *People*—also apply to Developer. Same applies for properties, where for example changes is annotated with the verbs *change(s)*, *modify*, *modifies*, and *edit(s)*.

In contrast to OWL classes and properties, where the annotations are encoded directly in SEON, fact extractors have to generate meaningful *rdfs:label* values for individuals. In most cases, this process is straightforward: for Java classes, fields, and methods, the Java identifier is taken, whereas for bug reports, the issue-key provided by the issue tracker (e.g., "IVY-123" for the issue #123 of the Apache Ivy project) serves as label.

Both, the annotations and *rdfs:labels* are key to the query approach that we discuss in Sect. 5.2. When entering queries, the nouns and verbs are used to provide guidance in composing questions, such as "Which Programmer modifies the method foo()?" or "What methods call bar()?". The natural language annotations of SEON

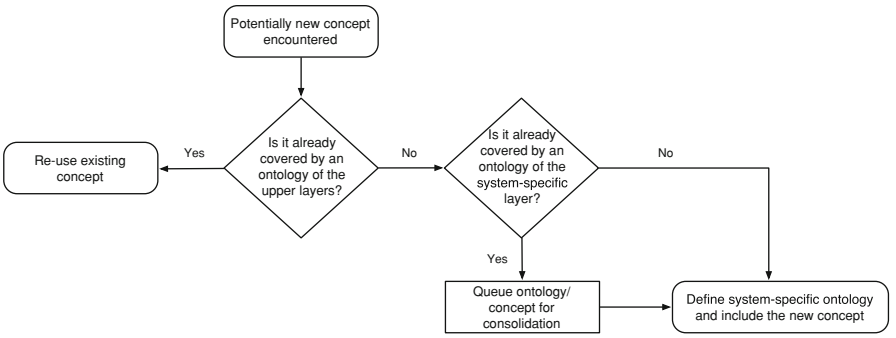

Fig. 3 Informal design process when encountering concepts during the conception of an analysis

also enhance some of the Web front-ends of the software analysis services presented in Sect. 5.1. The annotations are used to automate the generation of simple human-readable reports, e.g., "Michael Würsch commits Revisions 1–100". or "The class DBAccess has changed 50 times".

4.6 Our knowledge engineering process

Choosing which concepts should be included in an ontology in general, and assigning concepts to a layer of SEON in particular, is not always straight-forward. In the following we therefore briefly sketch the informal ontology design process used for SEON, which is illustrated in Fig. 3.

Knowledge engineers often start from an abstract high-level view when they identify and describe the important concepts in a domain under discourse. Then these concepts are iteratively validated and refined against the reality. In contrast to this top-down approach, we follow a more data-driven, bottom-up approach. At the beginning of the conception phase of a new software evolution support tool or data importer, we quickly model the important concepts of its domain, while neglecting those concepts that are not of immediate use for our purpose. For each important concept, we check whether it is already represented in one of SEON upper layers, e.g., the domain-specific layer, and re-use the existing concepts whenever possible. If the concept is not yet defined, we first stage the concept in a system-specific ontology for the specific system. Additionally, we check whether we have already defined similar concepts in other system-specific ontologies and, if so, queue them for consolidation. We usually post-pone the consolidation step until we reached a sufficient understanding of the problem domain—system-specific ontologies therefore act like an incubator to new concepts.

When we model, for example, the concepts of the two programming languages Java and C++, we first create two distinct system-specific ontologies. Then we compare the results and move the commonalities, such as Class, Field, Method, extends, invokes, etc., to SEON's domain-specific layer. The concepts that apply only to C++, such as Struct, Function Pointer, Header File, and the Java-exclusive concepts, e.g., Interface, Annotation, and Inner (Anonymous) Class, remain in the respective system-specific

ontologies. Pervasive concepts, i.e., those that apply to multiple domains, for example File, are promoted from the domain-specific to the domain-spanning—or even to the general layer of SEON.

4.7 An example scenario: clone evolution

Code clone detection in source code has been a lively field of research for many years now and it is generally accepted that duplicated code violates the *Don't Repeat Yourself (DRY)* principle [31], which can lead to software that is harder to maintain. An interesting aspect of code duplication is how clones evolve over time. This was, for example, investigated by Kim et al. [39].

Now consider the following scenario, where a researcher decides to carry out a similar study to the one presented by Kim et al. [39]. In particular, the researcher wants to find out whether the number and size of duplicated fragments change over the lifetime of a Java program. We assume that a clone detector was already selected; scripts to check-out every version of the source code files from an SVN repository have been developed. What is left, is to devise a tool that runs the clone detector on the data to perform the analysis. For that, the researcher needs to decide what meta-model should be used to represent the data under analysis, as well as the results of the analysis.

SEON provides all the necessary means to describe such knowledge. In the following, we briefly discuss how the relevant concepts and their relations are distributed over the four layers of our ontology pyramid. The OWL classes and object properties for the scenario are illustrated in Fig. 4. The illustration omits datatype properties for the sake of simplicity.

The core concept for this analysis is Clone. A clone belongsTo a CloneClass of duplicated fragments that are similar in syntax or semantics. While the concepts of our clone ontology might not suffice to represent all possible variants of clone analyses, it is straightforward to extend the existing ones. For example, one could specialize the concept Clone with different types of clones, such as SemanticClone or SyntacticClone to provide further classification. Or, additional object properties could link clones to issues for investigations on whether duplication leads to more bugs, and so on.

A Committer introduces a clone when she commits a new Version of a VersionedFile to the SVN repository. Committers are Developers that can check-in modifications. They are one of the many Stakeholders associated with the development process. Versioned files are Files managed by a version control system. Files are among the Artifacts that are produced when software is created. Clones occur in a particular CodeEntity, such as in a ComplexType (i.e., a class, interface, enum, etc.), a Method, etc. The size of such a piece of code, as well as the size of a clone, can be assessed by a Measurement. An adequate Measure for that is the number of lines of code, LOC.

The OWL classes Cloning and Commit are special cases: in principle, the relationship between clones and committers is already sufficiently stated by the object property introduces. However, in some cases, we also want to express that the introduction of a clone is an Activity with a certain time stamp and carried out by a particular stakeholder. There are two ways to do that. The first is reification, which allows for statements about statements. The second is to define an association class. Since reifi-

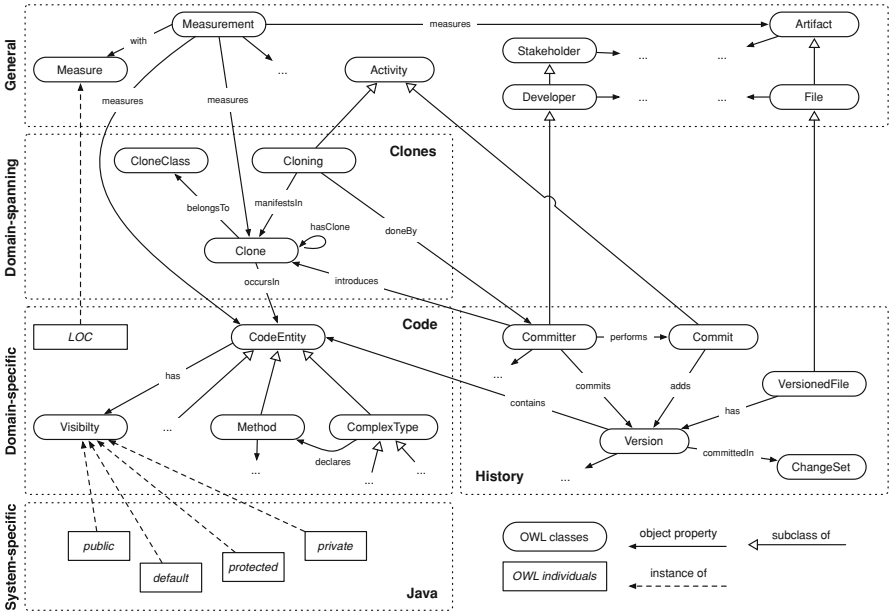

Fig. 4 The SEON concepts involved in a clone evolution analysis scenario

cation has not been widely adopted in the Semantic Web, we decided for the second variant and defined the OWL class Cloning to represent the introduction of a clone. A clone introduction is doneBy a committer and manifestsIn a new clone. A similar case is that of a Commit. It is also an activity that a committer performs and which adds a new version to a file. This apparent redundancy in the ontology definition allows us to support a wider range of applications. The query approach discussed in Sect. 5.2 works better with triples, such as "Committer$_A$ commits Version$_B$", that are close to the subject-predicate-objected sentence structure in English. On the other hand, the tool presented in Sect. 5.3 explicitly queries for activities to generate, e.g., timeline views. Fact extractors do not necessarily need to create both, an individual of Cloning and the statement "Committer$_X$ introduces Clone$_Y$". In many cases, we defined rules in the Semantic Web Rule Language (SWRL) [30], similar to the one in Listing 1. The rule states that, if some cloning activity has been carried out by a committer, and the cloning manifested in a clone, then the committer has introduced a new clone. With a reasoner, we can then automatically infer the missing triples for particular cases.

```
Cloning(?cloning), doneBy(?cloning, ?committer),
manifestsIn(?cloning, ?clone) → introduces(?committer, ?clone)
```

Listing 1 An Example for a SWRL rule defined by SEON

```
SELECT ?clone ?size ?version
WHERE
   { ?code         rdf:type        seon:CodeEntity
     ?clone        rdf:type        seon:Clone ;
                   seon:occursIn   ?code
     ?version      rdf:type        seon:Version ;
                   seon:contains   ?code
     ?measurement  rdf:type        seon:Measurement ;
                   seon:with       seon:LOC ;
                   seon:hasValue   ?size ;
                   seon:measures   ?clone }
```

Listing 2 SPARQL query returning Clones incl. size and version they appear in

Notable in Fig. 4 is also the OWL class Visibility. In most object-oriented programming languages, there exists an information-hiding mechanism to control the access of parts of the code. In Java, there are the visibility modifiers public, default, protected, and private that apply to types and their members. The actual instances of the visibility modifiers are defined in a system-specific (**Java**) ontology because there are quite significant differences in the meaning of such modifiers depending on the programming language used. The visibility concept, however, belongs to the domain-specific layer together with the other abstractions of **Code**. The layer also contains the predefined LOC individual, because the measure is clearly associated with program code. In our analysis scenario, there are no domain-spanning measures needed. The **History** ontology is located at the same level of abstraction as the **Code**. Currently, there are no system-specific extensions to it. The **Clones** ontology is domain-spanning—it relates to the **Code**, as well as to the **History**. The general concepts layer then provides abstractions for various concepts used in the lower layers.

Coming back to our initial clone evolution analysis scenario, we conclude that SEON provides the concepts necessary to support it. Once the ontology has been populated by a fact extractor, a concise SPARQL query can be issued to retrieve all clones, their size, and the versions they occur in. The query is given in Listing 2. Note that we have left out the prefix definition part of the query: the prefix *rdf* refers to http://www.w3.org/1999/02/22-rdf-syntax-ns#, whereas we assume that the *seon* prefix stands for http://se-on.org/ontologies/. In reality, each of the different layers of SEON has its own prefix/namespace.

5 Applications powered by SEON

In the following, we describe three different applications that work with SEON as their semantic backbone. The first one is our software evolution analysis web service platform SOFAS; the second one is HAWKSHAW, a natural language interface for answering program comprehension questions; and the third application is a recommender system called Semantic Visualization Broker (SVB). SVB analyzes the semantics of a given set of data and comes up with a list of visualizations that could be helpful to gain a deeper understanding of the software system under analysis. We have fully imple-

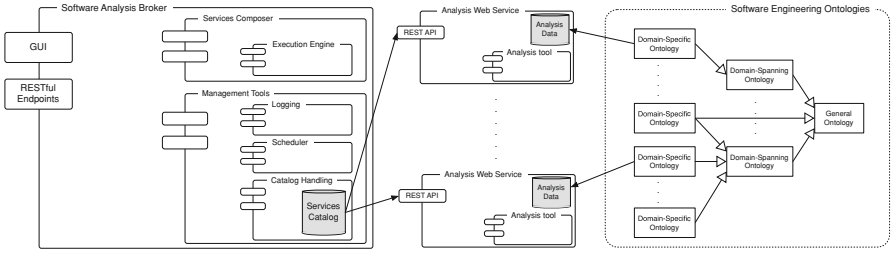

Fig. 5 The SOFAS architecture [20]

mented the three approaches in proof-of-concept tools. SOFAS and HAWKSHAW are even available for download on the SEON website.

5.1 Software analysis services

Mining Software Repositories has been an active field of research for many years, and various analysis techniques have been proposed, based on the idea that software engineers can learn from the development history of programs.

No matter whether these approaches are concerned with code analysis, code duplication, bug prediction, or any of the other repository-based analyses, many of them have in common that researchers had to build data extractors for version control repositories, issue trackers, mailing lists, and so on. While these efforts share many similarities, synergies are hard to exploit as many tools were designed to work stand-alone. The outcome is a diversity of platforms, similar, yet incompatible meta-models, and tool-specific input and output formats.

To overcome these challenges, we have devised SOFAS[11] (SOFtware Analysis Services), which we presented in [20]. SOFAS allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer (REST, as introduced by Fielding in [15]) around resources on the Web.

An overview on the architecture of SOFAS is given in Fig. 5. The architecture is made up by three main constituents: *Software Analysis Web Services*, a *Software Analysis Broker*, and *Software Analysis Ontologies* being part of SEON. The software analysis web services "wrap" already existing analysis tools by exposing their functionalities and data through standard RESTful web service interfaces. The broker acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. As such, the domain of analysis services is described in a semantical way enabling users to browse and search for their analysis service of interest. SEON defines and represents the data consumed and produced by the different services.

REST provides us a truly uniform interface to describe all the analysis services in the SOFAS architecture, the structure of their input and output, and how to invoke them at a syntactic level. However, there is no way to programmatically know what a service

[11] SOFAS is available online at http://se-on.org/sofas/.

actually offers and what the data means that it consumes and produces. Ontologies in general, and SEON in particular, help tackling both problems by providing meaningful service descriptions and data representation.

The Semantic Web leverages SOFAS in multiple ways. First, every resource gets a de-referenceable URI assigned. URIs align well with the REST principles and allow one service to hand-over artifacts to another one in a straight-forward manner. Next, the formal data semantics achieved with SEON helps in clearly specifying the input expected, as well as the output generated by the services, which increases interoperability and simplifies reuse of processing results. This is achieved by slightly expanding the Web Application Description Language (WADL) [24] with annotations inspired by SAWSDL (Semantic Annotations for WSDL) [14]. With them, the input and output of the services can be declared as being described by SEON. Last but not least, the footprint of the information exchanged by the services can be reduced by incorporating a reasoner. Only a limited set of triples then needs to be passed along by the sender and reasoning can be done by the receiver to add additional triples, if needed.

5.2 Supporting developers with natural language

In [59] we presented a framework for software engineers to answer common program comprehension questions with *guided-input natural language* queries, for example those questions presented by Silito et al. [50]. The framework is called HAWKSHAW[12] and has been implemented as a set of plug-ins for the Eclipse IDE. Figure 6 shows a screenshot of our query interface in action. In the example, a user has already started to compose a query. Three words have been typed in so far, "What Method invokes", and the drop-down menu presents the full list of methods that can be entered to complete the query.

The HAWKSHAW approach follows a method coined *Conceptual Authoring* or WYSIWYM (What You See Is What You Meant) by Hallet et al. [25] and Power et al. [48]. This means that, for composing queries, all editing operations are defined directly on an underlying logical representation, in our case SEON. However, the users do not need to know the underlying formalism because they are only exposed to a natural language representation of the ontology.

We use a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded [5]. The static part needs to be defined manually and, additionally, contains information on how to translate the user input into SPARQL. We generate the dynamic part from labels of the individuals, from the identifiers of the classes and properties, as well as from the SEON natural language annotations (see Sect. 4).

The static grammar basically defines a stub. In the example given above, the grammar describes that, after one of the interrogative determiners "What" or "Which", the

[12] Our tool is named after Hawkshaw the Detective, a comic strip popular in the first half of the twentieth century. *Hawkshaw* meant a detective in the slang of that time. The tool HAWKSHAW is available for download at http://se-on.org/hawkshaw/.

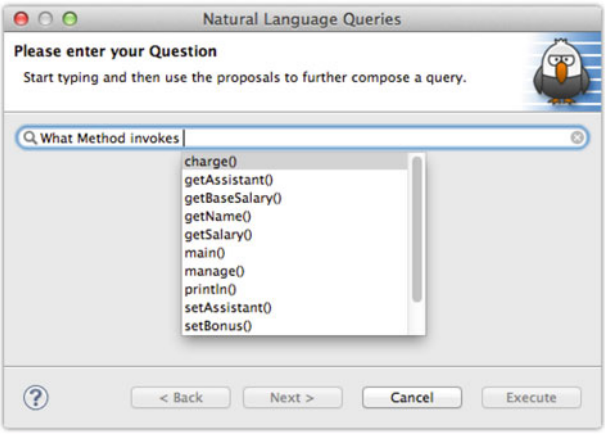

Fig. 6 The guided-input natural language interface powered by SEON

subject of the sentence needs to follow. The subject needs to be an OWL class defined by SEON. Further, the verb of the sentence has to be an object property that fits the subject, i.e., the object property has the class in its domain that has been selected as the sentence's subject. Object properties not fulfilling this constraint will not be presented to the user. Similarly the object of the sentence is an individual of a class in the ontology. The individual's class has to comply to the range specified for the object property, otherwise it will not be shown either. The stub provided by the static grammar then looks as follows: "What <class> <object-property> <individual>?"

The dynamic part of the grammar provides the replacements for the placeholders in the stub (denoted by < >). These replacements are presented to the user. Consider "What Method[1] invokes[2] charge()[3] ?". In this query, ([1]) is a label for the OWL class *JavaMethod*, ([2]) comes from the object property *invokesMethod*, and ([3]) from a human-readable label for one of the OWL individuals that have the class *JavaMethod*.

The utilization of the SEON ontologies for driving HAWKSHAW yields several major benefits: Ontologies are described in terms of triples of *subject*, *predicate*, and *object*. This structure strongly resembles how humans talk about things and can be easily transformed into natural language sentences. A surprisingly small set of static grammar rules allows for a variety of different queries.

Properties in OWL are a binary relation that can be restricted by specifying *domain* and *range*. In triples this means that the domain restricts the possible values of the subject and the range restricts the values of the object. For our query approach, this information can be exploited to filter the verbs that can follow a given subject, or the objects that can follow a given verb. For example the question "Which developer is assigned to issue #133?" makes sense, whereas "What field invokes class A?" does not.

We employ the Pellet reasoner [51] to infer specializations or generalizations. When we ask for, e.g.,"What persons are contributing to project X?", we are not only interested in a list of direct instances of the concept *Person*, but also in specializations, such as *Developers*, *Testers*, etc. Similarly, whenever we know that developers *create*

or *change* an artifact, we also want to generalize that they are *contributing* to the project. Reasoners greatly simplify data extraction, as they reduce the amount of explicit information that we need to state in our models.

5.3 Semantic Visualization Broker

The third application presented in this paper addresses the hardly known capabilities of software visualizations. The Semantic Visualization Broker (SVB) is essentially a recommender tool that suggests to the user suitable visualizations for a given set of data. The data can originate from the results of a query composed with the HAWKSHAW approach (Sect. 5.2), but also from a SOFAS analysis workflow (Sect. 5.1), or virtually any other source of RDF/OWL data.

Visualization plug-ins can register themselves with the SVB and specify the semantics of the data they can handle. The SVB expects as input a knowledge base and a result set. The result set should consist of the information a user asked for, whereas the knowledge base provides the context, in case that the SVB or a visualization has to query for additional data. The SVB then invokes a reasoner to infer abstractions from the result set and compares the outcome with the registration that the visualization plug-ins have provided. Any matches are presented to the user. The user can then select one or several recommendations from the list and the SVB will invoke and configure the visualizations automatically with the input data.

When the SVB receives a set of individuals as result set, it will query the knowledge base for their data properties and for object properties that link those individuals together. We currently support four different scenarios, which we describe in the following. An overview on the implemented visualization types is given in Fig. 7.

Hierarchies If the SVB detects a hierarchical relationship between the individuals in the result set, it will recommend a simple tree-like widget (which has been omitted from Fig. 7—it is similar to the widgets well-known from file system explorers) and a tree map visualization. If the selected individuals have a size measurement assigned (e.g., for files the *lines of code* metric), the SVB will configure the tree map to incorporate the size of each individual to calculate the layout.

Measurements If more than one individual has measurements assigned, then the SVB recommends a visualization based on Radar Charts. Each axis of the chart represents a certain type of measure. The number of axes that are displayed is limited; whenever more measures are available, some of them are chosen randomly and the user is given the possibility to reconfigure the selection. If measurements are available for more than one version of the individuals (e.g., for files under version control), then each axis will display multiple entries.

Activities In the case that most of the individuals represent an activity with a timestamp assigned, the SVB will automatically come up with a scrollable timeline-like visualization.

Miscellaneous data As a fallback, if none of the cases above apply, the broker will suggest a simple graph-based explorer that displays individuals and data values as

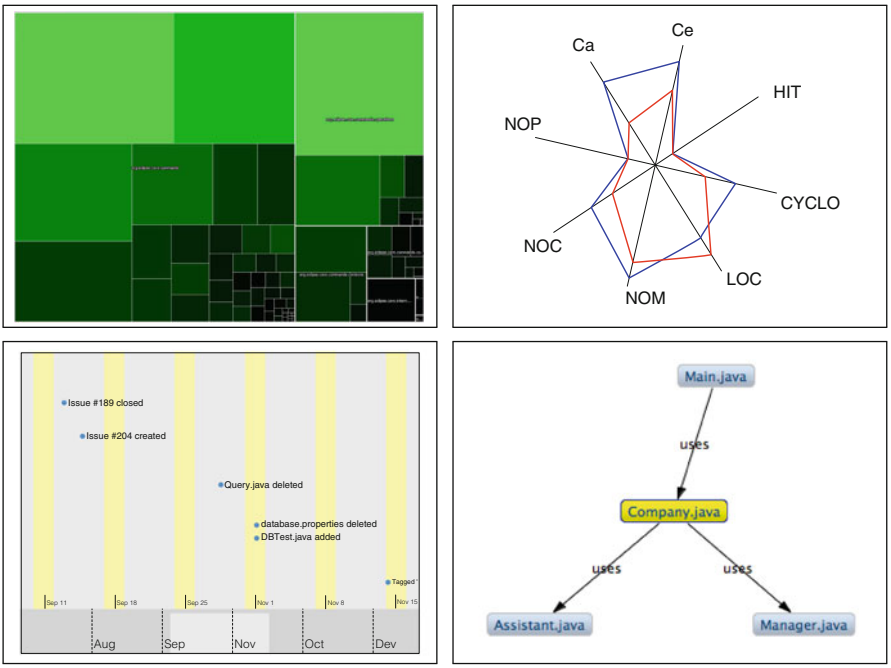

Fig. 7 The types of visualizations currently supported by the Semantic Visualization Broker: the *upper left* figure shows a tree map of a Java system, the *upper right* one shows a radar chart with measurements for two different versions of a Java class, the *lower left* figure shows a timeline with software evolution activities, and the *lower right* one shows a simple graph-based explorer displaying the dependencies among four Java classes

nodes and properties as edges. Unless the properties are defined as being *symmetric*, the corresponding edges will be directed.

Labels displayed in each of the visualizations are derived either from the RDF Schema labels or from the natural language annotations of SEON. The clear, machine-processable semantics of the data enable the SVB to make educated guesses on what visualizations may be appropriate. The power of a reasoner allows us to specify the concepts and relations supported by a visualization in a very generic way—the reasoner will automatically infer a hierarchical relationship from a set of triples containing, "Class$_A$ declaresMethod Method$_B$" and propose a tree-based visualization consequently.

The SVB offers quite some potential for enhancements. For example, we will explore the range of visualizations it can support and to what extent it is generalizable to non-visual applications.

6 Related work

In this section, we briefly sketch existing work involving ontologies in software engineering. We refrain from discussing publications that are only related to the approaches

presented in Sect. 5, but not particularly to the Semantic Web and ontologies. Related work in the context of software analysis services was already given in [20], whereas research in the area of program comprehension and developer support has been discussed extensively in [59].

A general overview of applications of ontologies in software engineering has been given in [23,27,55]. All of these publications promoted the theoretical benefits offered by different characteristics of ontologies, such as explicit semantics and taxonomy, lack of polysemy, ease of communication and automatic data exchange between distinct tools, and computational inference. In the following, we elaborate on how ontologies were applied to advance particular fields of research in software engineering. To the best of our knowledge, SEON is the only approach that describes software evolution data on multiple abstraction layers. Another unique selling proposition of our family of ontologies is that they were validated in three very distinct scenarios (cf. Sect. 5), whereas most other ontologies were deployed only in a rather specific environment.

6.1 Ontologies for software artifacts

Different approaches to establish taxonomies for software engineering by means of ontologies have been presented recently.

Hyland-Wood et al. [32] proposed an OWL ontology of software engineering concepts, including classes, tests, metrics, and requirements. Bertoa et al. focused on software measurement [6]. Their software measurement ontology influenced the respective concepts of SEON. Bertoa et al.'s set of measurement concepts is more complete, whereas our ontology focuses on simplicity.

Oberle et al. [45] recognized that *the domain of software is a primary candidate for being formalized in an ontology*, being both, sufficiently complex and reasonably stable in paradigms and aspects. Consequently, a reference ontology for software was presented to distinguish fundamental concepts in the domain of software engineering, such as data and software.

These three approaches show some overlap in concepts with our ontologies but they neglected evolutionary aspects, whereas SEON explicitly models the development history of software systems, such as versions, releases, bugs, etc.

6.2 Ontologies for software maintenance

Several approaches relied on ontologies to support software maintenance—be it to describe domain knowledge of developers, source code and documentation to support program comprehension, or to infer bugs based on a set of heuristics.

LaSSIE, presented by Devanbu et al. [11], was an early attempt to integrate multiple views on a software system in a knowledge base. It also provided semantic retrieval through a natural language interface. Frame systems, a conceptual predecessor to the ontologies of the Semantic Web, were used to encode the knowledge. The main goal of LaSSIE was to preserve knowledge of the application domain for maintainers of the software system.

The author of [56] found LaSSIE's source code model too course-grained and not applicable to object-oriented code. Therefore, he augmented abstract syntax trees with semantics. For that DL was used to develop an ontology for software understanding. The ontology, in combination with an inferencer, then enabled automatic detection of side effects in code and path-tracing.

Witte et al. [58] used text mining and static code analysis to map documentation to source code for software maintenance purposes. These mappings were represented in RDF.

Yu et al. [61] also represented static source code information by means of an OWL ontology. They further used the Semantic Web Rule Language (SWRL) [30] to describe common bugs found in code. With a rule engine, inference results could be obtained to indicate the presence of bugs.

Our natural language query approach HAWKSHAW described in Sect. 5.2 shares many similarities with LaSSIE but, thanks to SEON, potentially covers a broader range of concepts. However, SEON does not incorporate application-specific knowledge. The other three approaches described above focus only on source code, whereas we incorporate many different artifacts, stakeholders, and their activities.

6.3 Ontologies for software reuse

Properties of software components have been represented with ontologies in the past. Such properties ranged from programming languages and source code facts to licenses, software types and application domains. The common goal was to foster reuse by enabling searches in a component database for certain criteria that relate to, e.g., particular requirements.

Happel et al. [26] proposed various ontologies to foster software reuse. In their KOntoR approach, they provided background knowledge about software artifacts, such as the programming language used or licensing models. The artifacts, along with their ontology meta-data, were stored in a query-able central repository to facilitate reuse.

The authors of [28] used ontologies to describe software components. They classified software with respect to a hierarchy of software types. An example given in their paper was IBM's DB2, which is a relational database management system (RDBMS); RDBMSs were then considered as a subclass of database managements systems, and so on. The authors additionally defined hierarchies of functionality types (e.g., *importing data* as a special kind of *adding data*) to further describe the features of components. An algorithm was presented to automatically find an optimal component solution for a given set of requirements.

Dietrich and Elgar [12] developed a tool that scans the abstract syntax tree of Java programs and detects design patterns for documentation purposes. The design patterns were described in terms of OWL ontologies.

Alnusair and Zhao [1], similar to Hartig et al., used OWL ontologies for component descriptions. They took a three-layered approach for their ontological descriptions: an ontology representing static source code information, different domain ontologies to conceptualize the domain of each component (e.g., finance or medicine), and an ontology that extended their source code ontology with component-related concepts. The

authors supported several kinds of query methods against their component knowledge base: type or signature-based queries, meta-data keyword queries, or pure semantic-based queries.

SEON, in contrast to these four approaches, does neither model software systems at a component level, nor does it represent design patterns. However, in our ontologies, we model other important facets of software that could yield interesting synergies when synthesized with these ontologies for software reuse, for example, to give insights on the maintainability of particular components. This could help software engineers to make even more profound decisions on what components their software systems should be based on.

6.4 Ontologies in search-driven software engineering

The field of search-driven software engineering has produced various code search engines. Some of them simply use OWL/RDF as an internal representation of program code and allow users to issue SPARQL queries against the code base [35]. Others exploit the possibilities of the Semantic Web further. Durão et al. [13], for example, classified source code according to domains, such as Graphical User Interfaces, I/O, Networking, Security, and so on. The authors then provided a keyword search over the code base, and the results of the queries could be limited to return only matches from a particular domain.

The applications of SEON presented in Sect. 5 also make extensive use of the Semantic Web's search facilities, in particular of SPARQL. Source code search, however, is not the main purpose of our applications but rather a means to an end. Nevertheless, it is easily conceivable that we might adopt a code search engine as a SOFAS service in the future.

6.5 Ontologies in mining software repositories

Several researchers have described software evolution artifacts found in software repositories with OWL ontologies. Their approaches integrated different artifact sources to facilitate common repository mining activities. The flexible RDF data model, automatic semantic mashup technologies, and the powerful search-facilities of the Semantic Web have proven their use in this context.

Tappolet made a case for incorporating Semantic Web technology in software repositories in [53]. The authors claimed that this would greatly facilitate the handling of distributed and heterogeneous software project data. Tappolet then presented a roadmap towards such semantics-aware software project repositories consisting of three main steps: (1) data representation by means of RDF/OWL ontologies, (2) intra-project repository integration, and finally (3) inter-project repository integration.

Based on these ideas, Kiefer et al. [37] presented EvoOnt, a software repository data exchange format based on OWL. EvoOnt involved three sub-ontologies: a software ontology model, a bug ontology model, and a version ontology model. The authors used a modified version of SPARQL to detect bad code smells, calculate metrics, and to extract data for visualizing changes in code over time. A reasoner was incorporated to

detect orphan methods, i.e., methods never called by any other methods in the system. Tappolet et al. recently extended the EvoOnt approach. Several software evolution analysis experiments from previous Mining Software Repositories Workshops were repeated and it was demonstrated by the authors that, if the data used for analysis were available in EvoOnt, then the analyses in 75 % of the selected MSR papers could be reduced to one or at most two simple SPARQL queries.

Iqbal et al. [33] discussed different scenarios and use cases for Linked Data in software engineering. They presented their *Linked Data Driven Software Development* (LD2SD) methodology, which involves transformation of software repository data into the RDF format and then indexing with a semantic indexer. The overall goal was to provide a uniform and central RDF-based access to JIRA bug trackers, Subversion, developer blogs, project mailing lists, etc. Integration between the repositories was achieved with *Semantic Pipes*, an RDF-based mashup technology. The results were finally injected into the DOM of a Web page (e.g., that of a bug tracker) to provide developers with additional, context-related information.

None of these approaches organize their ontologies in consecutive layers of abstractions with clear representational purpose, as we did for SEON. Instead, the authors have laid out their ontologies at a particular level of abstraction. For example, while most concepts in EvoOnt can be mapped 1:1 to concepts in SEON, it is conceptually situated somewhere between SEON's system- and domain-specific layers and lacks the domain-spanning and general concepts that we have defined.

Despite these limitations, we can envision interesting interactions between our semantics-aware applications and the technologies presented by the other authors. The SPARQL extension presented by Kiefer et al., for example, adds machine learning algorithms (SPARQL-ML [38]) and similarity joins (iSPARQL [36]) to the Semantic Web. Both extensions could lead to a complete new family of SOFAS services or at least simplify the implementation of existing ones. The semantic mashup technology used in LD2SD could further improve the presentation of the analysis results of our services.

7 Conclusions

Some decades ago, a team of developers could write industrial-strength software on their own, only with the aid of a simple text editor, a compiler, and perhaps a debugger. The software engineering landscape has changed dramatically since then.

Development teams have grown to dozens, and sometimes even hundreds of people. A plethora of tools have found their way into integrated development environments—without the help of these IDEs, we as programmers can barely imagine to write a single line of code anymore. Software repositories, such as version control systems and bug trackers, foster collaboration and provide means to control and reflect on the development processes.

With the increase in team size and tool support, the amount of data that breaks in on individual developers has grown to a point where it becomes harder and harder for them to grasp implicit relationships among artifacts stored in different locations. Too much time is lost in distinguishing useful information from random noise. In

consequence, software engineers are hardly able to fully exploit all their tooling and productivity gains are thus wasted. A new generation of tools is therefore needed—tools that can make use of the semantics of the underlying data to automate tedious processes and filter irrelevant information. The Semantic Web provides a framework to build such tools.

In this paper, we have investigated the research question how software evolution knowledge can be adequately represented by means of ontologies. As an answer to this question, we presented SEON, a family of ontologies that describe many different facets of a software's life-cycle. SEON is unique in that it is comprised of multiple abstraction layers. Our ontologies provide a shared taxonomy of important software engineering concepts and already have found multiple applications. Three of them were discussed in this paper, and we argued that each application clearly benefits from the use of Semantic Web technologies. SOFAS, our software analysis services platform, used SEON as a formal description of the input and output of its individual services. Our guided-input natural language approach HAWKSHAW exploited the clear semantics of OWL to translate program comprehension questions formulated by developers in quasi-natural language to the formal Semantic Web query language SPARQL. This was possible, since the natural language annotation layer of SEON bridged the gap between machine-processable and human-understandable knowledge. SVB, our Semantic Visualization Broker, relied on reasoning and explicit relations to automatically infer suitable visualizations for given sets of data. All of these three applications would have been significantly harder to implement without SEON and the use of Semantic Web technologies.

We only have started to exploit the potential that the Semantic Web could bring for software evolution support. Other researchers have begun to explore the opportunities and we hope that this paper can encourage even more to do so. A next important step is to consolidate other existing ontologies and to come up with layers of abstraction, similar to what we did with SEON. Based on this, software repositories need to be devised that are semantics-aware, i.e., that produce and consume data in the RDF/OWL format, and that expose stable de-referenceable URIs on the Web. When this is achieved, software repositories could ultimately blend into a queryable global information space of interlinked software evolution data.

Acknowledgments The authors would like to thank Emanuel Giger and the anonymous reviewers for their insightful comments.

References

1. Alnusair A, Zhao T (2011) Retrieving reusable software components using enhanced representation of domain knowledge. In: Recent trends in information reuse and integration. Springer, Wien
2. Ball T, Kim J, Porter A, Siy H (1997) If your version control system could talk. In: Proceedings of international workshop on process modelling and Empir Studies Softw Eng
3. Berners-Lee T, Fielding R, Masinter L (1998) RFC 2396: uniform resource identifiers (URI). IETF RFC. http://www.ietf.org/rfc/rfc2396.txt
4. Berners-Lee T, Hendler J, Lassila O (2001) The Semantic Web. Sci Am 284(5):34–43
5. Bernstein A, Kaufmann E, Kaiser C, Kiefer C (2006) Ginseng: a guided input natural language search engine for querying ontologies. In: Jena User Conference

6. Bertoa M, Vallecillo A, Garcia F (2006) An ontology for software measurement. In: Ontologies for software engineering and software technology. Springer, Heidelberg
7. Bevan J, E James Whitehead J, Kim S, Godfrey MW (2005) Facilitating software evolution research with Kenyon. In: Proceedings of joint European software engineering conference and symposium on foundations of software engineering, pp 177–186
8. Chen YF, Gansner ER, Koutsofios E (1998) A C++ data model supporting reachability analysis and dead code detection. Trans Softw Eng 24(9):682–694. doi:10.1109/32.713323
9. D'Ambros M, Gall HC, Lanza M, Pinzger M (2008) Analyzing software repositories to understand software evolution. In: Software evolution. Springer, Heidelberg
10. Dean M, Schreiber G (eds) (2004) OWL Web ontology language reference. W3C recommendation. http://www.w3.org/TR/owl-ref/
11. Devanbu P, Brachman R, Selfridge PG (1991) Lassie: a knowledge-based software information system. Commun ACM 34(5):34–49. doi:10.1145/103167.103172
12. Dietrich J, Elgar C (2005) A formal description of design patterns using owl. In: Proceedings of Australian software engineering conference. doi:10.1109/ASWEC.2005.6
13. Durão FA, Vanderlei TA, Almeida ES, de L Meira SR (2008) Applying a semantic layer in a source code search tool. In: Proceedings of symposium on applied computing. doi:10.1145/1363686.1363952
14. Farrell J, Lausen H (2007) Semantic annotations for WSDL and XML schema. W3C recommendation. http://www.w3.org/TR/sawsdl/
15. Fielding RT (2000) Architectural styles and architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine
16. Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: Proceedings of international conference software maintenance, pp 23–32
17. Fluri B, Würsch M, Pinzger M, Gall H (2007) Change distilling: tree differencing for fine-grained source code change extraction. Trans Softw Eng 33(11):725–743. doi:10.1109/TSE.2007.70731
18. Gall H, Hajek K, Jazayeri M (1998) Detection of logical coupling based on product release history. In: Proceedings of international conference software maintenance
19. Gall HC, Fluri B, Pinzger M (2009) Change analysis with Evolizer and ChangeDistiller. Software 26(1):26–33
20. Ghezzi G, Gall H (2011) SOFAS: a lightweight architecture for software analysis as a service. In: Working conference software architecture, pp 93–102. doi:10.1109/WICSA.2011.21
21. Giger E, Pinzger M, Gall H (2010) Predicting the fix time of bugs. In: Proceedings of international workshop on recommendation system for software engineering. doi:10.1145/1808920.1808933
22. Gruber TR (1993) A translation approach to portable ontology specifications. Knowl Acquis 5(2): 199–220. doi:10.1006/knac.1993.1008
23. Gruninger M, Lee J (2002) Ontology applications and design. Commun ACM 45(2):39–41
24. Hadley MJ (2009) Web application description language (wadl). W3C Member Submission. http://www.w3.org/Submission/wadl/
25. Hallett C, Scott D, Power R (2007) Composing questions through conceptual authoring. Comput Linguist 33:105–133. doi:10.1162/coli.2007.33.1.105
26. Happel H, Korthaus A, Seedorf S, Tomczyk P (2006) KOntoR: an ontology-enabled approach to software reuse. In: Proceedings of international conference on software engineering and Knowledge engineering
27. Happel HJ, Seedorf S (2006) Applications of ontologies in software engineering. In: Proceedings of international workshop on semantic Web enabled software engineering
28. Hartig O, Kost M, Freytag JC (2008) Automatic component selection with semantic technologies. In: Proceedings of international workshop on semantic Web enabled software engineering
29. Hespos SJ, Spelke ES (2004) Conceptual precursors to language. Nature 430(6998):453–456
30. Horrocks I, Patel-Schneider PF, Boley H, Tabet S, Grosof B, Dean M (2004) SWRL: a semantic web rule language combining OWL and RuleML. W3C Member Submission. http://www.w3.org/Submission/SWRL/
31. Hunt A, Thomas D (1999) The pragmatic programmer: from journeyman to master. Addison-Wesley, Boston
32. Hyland-Wood D, Carrington D, Kaplan S (2006) Toward a software maintenance methodology using semantic web techniques. In: Proceedings of international workshop on software evolution, pp 23–30. doi:10.1109/SOFTWARE-EVOLVABILITY.2006.16

33. Iqbal A, Ureche O, Hausenblas M, Tummarello G (2009) LD2SD: linked data driven software development. In: Proceedings of international conference on software engineering and knowledge engineering
34. Kagdi H, Collard ML, Maletic JI (2007) A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J Softw Maint Evol 19:77–131. doi:10.1002/smr.344
35. Keivanloo I, Roostapour L, Schugerl P, Rilling J (2010) Semantic Web-based source code search. In: Proceedings of international workshop on semantic Web enabled software engineering
36. Kiefer C, Bernstein A, Stocker M (2007a) The fundamentals of iSPARQL: a virtual triple approach for similarity-based semantic web tasks. In: Proceedings of international conference on semantic Web and Asian semantic Web conference
37. Kiefer C, Bernstein A, Tappolet J (2007b) Mining software repositories with iSPAROL and a software evolution ontology. In: Proceedings of international workshop on mining software repositories. doi:10.1109/MSR.2007.21
38. Kiefer C, Bernstein A, Locher A (2008) Adding data mining support to SPARQL via statistical relational learning methods. In: Proceedings of European semantic Web conference
39. Kim M, Sazawal V, Notkin D, Murphy G (2005) An empirical study of code clone genealogies. In: Proceedings of joint European software engineering conference and symposium on foundations of software engineering, doi:10.1145/1081706.1081737
40. Klyne G, Carroll JJ (eds) (2004) Resource description framework (RDF): concepts and abstract syntax. W3C Recommendation. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/
41. Kupershmidt I, Su QJ, Grewal A, Sundaresh S, Halperin I, Flynn J, Shekar M, Wang H, Park J, Cui W, Wall GD, Wisotzkey R, Alag S, Akhtari S, Ronaghi M (2010) Ontology-based meta-analysis of global collections of high-throughput public data. PLoS ONE 5(9). doi:10.1371/journal.pone.0013066
42. Lanza M, Marinescu R, Ducasse S (2005) Object-oriented metrics in practice. Springer, Heidelberg
43. Lethbridge TC, Tichelaar S, Plödereder E (2004) The Dagstuhl middle metamodel: a schema for reverse engineering. Electron Notes Theor Comput Sci 94:7–18
44. Müller HA, Klashinsky K (1988) Rigi-a system for programming-in-the-large. In: Proceedings of international conference on software engineering
45. Oberle D, Grimm S, Staab S (2009) An ontology for software. In: Handbook on ontologies in information systems, 2nd edn. Springer, Heidelberg. doi:10.1007/978-3-540-92673-3
46. Object Management Group (1998) XML metadata interchange (XMI). Technical Report OMG Document ad/98-10-05
47. Patel-Schneider PF, Hayes P, Horrocks I (eds) (2004) OWL Web ontology language semantics and abstract syntax. W3C Recommendation. http://www.w3.org/TR/owl-semantics/
48. Power R, Scott D, Evans R (1998) What you see is what you meant: direct knowledge editing with natural language feedback. In: Proceedings biennial European conference on artificial intelligence, pp 675–681
49. Prud'hommeaux E, Seaborne A (eds) (2008) SPARQL query language for RDF. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/
50. Sillito J, Murphy GC, De Volder K (2006) Questions programmers ask during software evolution tasks. In: Proceedings of international symposium on foundations of software engineering, pp 23–34. doi:10.1145/1181775.1181779
51. Sirin E, Parsia B, Grau B, Kalyanpur A, Katz Y (2007) Pellet: a practical OWL-DL reasoner. J Web Semant 5(2):51–53. doi:10.1016/j.websem.2007.03.004
52. Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of international workshop on mining software repositories, pp 1–5. doi:10.1145/1083142.1083147
53. Tappolet J (2008) Semantics-aware software project repositories. In: Proceedings of the ESWC'08 Ph.D. Symposium
54. Tichelaar S, Ducasse S, Demeyer S (2000) FAMIX and XMI. In: Proceedings of working conference on reverse engineering, p 296
55. Uschold M, Jasper R (1996) A framework for understanding and classifying ontology applications. In: Proceedings of international workshop on ontology and problem solving methods
56. Welty CA (1997) Augmenting abstract syntax trees for program understanding. In: Proceedings of international conference on automated software engineering. doi:10.1109/ASE.1997.632832
57. Winter A, Kullbach B, Riediger V (2002) An overview of the GXL graph exchange language. In: Diehl S (ed) Software visualization. Springer, Heidelberg, pp 324–336
58. Witte R, Zhang Y, Rilling J (2007) Empowering software maintainers with semantic web technologies. In: Proceedings of European semantic Web conference. doi:10.1007/978-3-540-72667-8_5

59. Würsch M, Ghezzi G, Reif G, Gall HC (2010a) Supporting developers with natural language queries. In: Proceedings of international conference on software engineering, pp 165–174. doi:10.1145/1806799.1806827
60. Würsch M, Reif G, Demeyer S, Gall HC (2010b) Fostering synergies: how semantic web technology could influence software repositories. In: Proceedings of international workshop on search-driven software development, pp 45–48. doi:10.1145/1809175.1809187
61. Yu L, Zhou J, Yi Y, Li P, Wang Q (2008) Ontology model-based static analysis on java programs. Comput Softw Appl. doi:10.1109/COMPSAC.2008.73